[Skip to main content](https://www.bezkoder.com/spring-boot-angular-13-crud/#content)

[**BezKoder**](https://www.bezkoder.com/)

* [Courses](https://www.bezkoder.com/category/courses/)
* [Full Stack](https://www.bezkoder.com/category/full-stack/)
* [Spring](https://www.bezkoder.com/category/spring/)
* [Node](https://www.bezkoder.com/node-js-tutorial/)
* [Vue](https://www.bezkoder.com/category/vue/)
* [Angular](https://www.bezkoder.com/category/angular/)
* [React](https://www.bezkoder.com/category/react/)
* [Firebase](https://www.bezkoder.com/category/firebase/)
* [Django](https://www.bezkoder.com/category/django/)
* [Dart](https://www.bezkoder.com/category/dart/)
* [JsonFormatter](https://www.bezkoder.com/json-formatter/)
* [Categories](https://www.bezkoder.com/categories/)

**Spring Boot + Angular 13: CRUD example (full stack)**

[Last modified: June 23, 2022](https://www.bezkoder.com/spring-boot-angular-13-crud/)  [bezkoder](https://www.bezkoder.com/author/bezkoder/)  [Angular](https://www.bezkoder.com/category/angular/), [Full Stack](https://www.bezkoder.com/category/full-stack/), [Spring](https://www.bezkoder.com/category/spring/)

In this tutorial, we will learn how to build a full stack Spring Boot + Angular 13 example with a CRUD Application. The back-end server uses Spring Boot with Spring Web MVC for REST Controller and Spring Data JPA for interacting with embedded database (H2 database). Front-end side is made with Angular 13, HttpClient, Router and Bootstrap 4.

Run both Project on same server/port:  
[How to Integrate Angular with Spring Boot Rest API](https://bezkoder.com/integrate-angular-12-spring-boot/)

Security: [Angular + Spring Boot: JWT Authentication example](https://www.bezkoder.com/angular-13-spring-boot-jwt-auth/)  
Upload: [Angular + Spring Boot: File upload/download example](https://www.bezkoder.com/angular-13-spring-boot-file-upload/)  
Pagination: [Angular + Spring Boot: Pagination example](https://bezkoder.com/pagination-spring-boot-angular-12/)

Newer version: [Spring Boot + Angular 14: CRUD example](https://www.bezkoder.com/spring-boot-angular-14-crud/)

**Contents**[[hide](https://www.bezkoder.com/spring-boot-angular-13-crud/)]

* [Spring Boot and Angular 13 CRUD example](https://www.bezkoder.com/spring-boot-angular-13-crud/#Spring_Boot_and_Angular_13_CRUD_example)
* [Spring Boot + Angular 13 fullstack Architecture](https://www.bezkoder.com/spring-boot-angular-13-crud/#Spring_Boot_Angular_13_fullstack_Architecture)
* [Video](https://www.bezkoder.com/spring-boot-angular-13-crud/#Video)
* [Spring Boot Back-end](https://www.bezkoder.com/spring-boot-angular-13-crud/#Spring_Boot_Back-end)
  + [Overview](https://www.bezkoder.com/spring-boot-angular-13-crud/#Overview)
  + [Technology](https://www.bezkoder.com/spring-boot-angular-13-crud/#Technology)
  + [Project Structure](https://www.bezkoder.com/spring-boot-angular-13-crud/#Project_Structure)
  + [Create & Setup Spring Boot project](https://www.bezkoder.com/spring-boot-angular-13-crud/#Create_038_Setup_Spring_Boot_project)
  + [Configure Spring Datasource, JPA, Hibernate](https://www.bezkoder.com/spring-boot-angular-13-crud/#Configure_Spring_Datasource_JPA_Hibernate)
  + [Define Data Model](https://www.bezkoder.com/spring-boot-angular-13-crud/#Define_Data_Model)
  + [Create Repository Interface](https://www.bezkoder.com/spring-boot-angular-13-crud/#Create_Repository_Interface)
  + [Create Spring Rest APIs Controller](https://www.bezkoder.com/spring-boot-angular-13-crud/#Create_Spring_Rest_APIs_Controller)
  + [Run the Spring Boot Server](https://www.bezkoder.com/spring-boot-angular-13-crud/#Run_the_Spring_Boot_Server)
* [Angular 13 Front-end](https://www.bezkoder.com/spring-boot-angular-13-crud/#Angular_13_Front-end)
  + [Overview](https://www.bezkoder.com/spring-boot-angular-13-crud/#Overview-2)
  + [Technology](https://www.bezkoder.com/spring-boot-angular-13-crud/#Technology-2)
  + [Project Structure](https://www.bezkoder.com/spring-boot-angular-13-crud/#Project_Structure-2)
  + [Setup Angular 13 Project](https://www.bezkoder.com/spring-boot-angular-13-crud/#Setup_Angular_13_Project)
  + [Set up App Module](https://www.bezkoder.com/spring-boot-angular-13-crud/#Set_up_App_Module)
  + [Define Routes for Angular AppRoutingModule](https://www.bezkoder.com/spring-boot-angular-13-crud/#Define_Routes_for_Angular_AppRoutingModule)
  + [Define Model Class](https://www.bezkoder.com/spring-boot-angular-13-crud/#Define_Model_Class)
  + [Create Data Service](https://www.bezkoder.com/spring-boot-angular-13-crud/#Create_Data_Service)
  + [Create Angular 13 Components](https://www.bezkoder.com/spring-boot-angular-13-crud/#Create_Angular_13_Components)
  + [Run the Angular 13 App](https://www.bezkoder.com/spring-boot-angular-13-crud/#Run_the_Angular_13_App)
* [Further Reading](https://www.bezkoder.com/spring-boot-angular-13-crud/#Further_Reading)
* [Source Code](https://www.bezkoder.com/spring-boot-angular-13-crud/#Source_Code)
* [Conclusion](https://www.bezkoder.com/spring-boot-angular-13-crud/#Conclusion)

**Spring Boot and Angular 13 CRUD example**

We will build a full-stack Angular 13 + Spring Boot Tutorial CRUD Application in that:

* Each Tutorial has id, title, description, published status.
* We can create, retrieve, update, delete Tutorials.
* We can also find Tutorials by title.

The images below shows screenshots of our System.

– Create a new Tutorial:

– Retrieve Tutorials:

– Click on **Edit** button to update a Tutorial:

On this Page, you can:

* change status to **Published** using **Publish** button
* remove the Tutorial from Database using **Delete** button
* update the Tutorial details on Database with **Update** button

If you want to implement Form Validation, please visit:  
[Angular Form Validation example (Reactive Forms)](https://www.bezkoder.com/angular-13-form-validation/)

– Search Tutorials by title:

**Spring Boot + Angular 13 fullstack Architecture**

Now look at the application architecture we will build:

– Spring Boot exports REST Apis using Spring Web MVC & interacts with embedded H2 Database using Spring Data JPA.  
– Angular 13 Client sends HTTP Requests and retrieve HTTP Responses using HttpClient Module, shows data on the components. We also use Angular Router for navigating to pages.

You can also find the Spring Restful Apis that works with other databases here:  
– [Spring JPA + PostgreSQL](https://bezkoder.com/spring-boot-postgresql-example/)  
– [Spring JPA + MySQL](https://bezkoder.com/spring-boot-jpa-crud-rest-api/)  
– [Spring Data + MongoDB](https://bezkoder.com/spring-boot-mongodb-crud/)  
– [Spring JPA + SQL Server](https://www.bezkoder.com/spring-boot-sql-server/)  
– [Spring JPA + Oracle](https://bezkoder.com/spring-boot-hibernate-oracle/)  
– [Spring Data + Cassandra](https://bezkoder.com/spring-boot-cassandra-crud/)

**Video**

This is our Angular + Spring Boot CRUD application demo and brief instruction:

In the video, we use Angular 10 with MySQL database, but the logic and UI are the same as this Angular version 13 and embedded database.

**Spring Boot Back-end**

**Overview**

These are APIs that Spring Boot App will export:

| **Methods** | **Urls** | **Actions** |
| --- | --- | --- |
| POST | /api/tutorials | create new Tutorial |
| GET | /api/tutorials | retrieve all Tutorials |
| GET | /api/tutorials/:id | retrieve a Tutorial by :id |
| PUT | /api/tutorials/:id | update a Tutorial by :id |
| DELETE | /api/tutorials/:id | delete a Tutorial by :id |
| DELETE | /api/tutorials | delete all Tutorials |
| GET | /api/tutorials?title=[keyword] | find all Tutorials which title contains keyword |

– We make CRUD operations & finder methods with Spring Data JPA’s JpaRepository.  
– The database will be H2 Database (in memory or on disk) by configuring project dependency & datasource.

**Technology**

* Java 8
* Spring Boot 2 (with Spring Web MVC, Spring Data JPA)
* H2 Database
* Maven 3.6.1

**Project Structure**

– Tutorial data model class corresponds to entity and table *tutorials*.  
– TutorialRepository is an interface that extends [JpaRepository](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) for CRUD methods and custom finder methods. It will be autowired in TutorialController.  
– TutorialController is a [RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) which has request mapping methods for RESTful requests such as: *getAllTutorials*, *createTutorial*, *updateTutorial*, *deleteTutorial*, *findByPublished*…  
– Configuration for Spring Datasource, JPA & Hibernate in **application.properties**.  
– **pom.xml** contains dependencies for Spring Boot and H2 Database.

**Create & Setup Spring Boot project**

Use [Spring web tool](https://start.spring.io/) or your development tool ([Spring Tool Suite](https://spring.io/tools), Eclipse, [Intellij](https://www.jetbrains.com/idea/download/)) to create a Spring Boot project.

Then open **pom.xml** and add these dependencies:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

**Configure Spring Datasource, JPA, Hibernate**

Under **src**/**main**/**resources** folder, open *application.properties* and write these lines.

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto= update

spring.h2.console.enabled=true

# default path: h2-console

spring.h2.console.path=/h2-ui

* spring.datasource.url: jdbc:h2:mem for In-memory database and jdbc:h2:file for disk-based database.
* spring.datasource.username & spring.datasource.password properties are the same as your database installation.
* Spring Boot uses Hibernate for JPA implementation, we configure H2Dialect for H2 Database
* spring.jpa.hibernate.ddl-auto is used for database initialization. We set the value to update value so that a table will be created in the database automatically corresponding to defined data model. Any change to the model will also trigger an update to the table. For production, this property should be validate.
* spring.h2.console.enabled=true tells the Spring to start H2 Database administration tool and you can access this tool on the browser: http://localhost:8080/h2-console.
* spring.h2.console.path=/h2-ui is for H2 console’s url, so the default url http://localhost:8080/h2-console will change to http://localhost:8080/h2-ui.

**Define Data Model**

Our Data model is Tutorial with four fields: id, title, description, published.  
In **model** package, we define Tutorial class.

*model/Tutorial.java*

package com.bezkoder.spring.datajpa.model;

import javax.persistence.\*;

@Entity

@Table(name = "tutorials")

public class Tutorial {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

@Column(name = "title")

private String title;

@Column(name = "description")

private String description;

@Column(name = "published")

private boolean published;

public Tutorial() {

}

...

}

– @Entity annotation indicates that the class is a persistent Java class.  
– @Table annotation provides the table that maps this entity.  
– @Id annotation is for the primary key.  
– @GeneratedValue annotation is used to define generation strategy for the primary key. GenerationType.AUTO means Auto Increment field.  
– @Column annotation is used to define the column in database that maps annotated field.

**Create Repository Interface**

Let’s create a repository to interact with Tutorials from the database.  
In **repository** package, create TutorialRepository interface that extends JpaRepository.

*repository/TutorialRepository.java*

package com.bezkoder.spring.datajpa.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.bezkoder.spring.datajpa.model.Tutorial;

public interface TutorialRepository extends JpaRepository<Tutorial, Long> {

List<Tutorial> findByPublished(boolean published);

List<Tutorial> findByTitleContaining(String title);

}

Now we can use JpaRepository’s methods: save(), findOne(), findById(), findAll(), count(), delete(), deleteById()… without implementing these methods.

We also define custom finder methods:  
– findByPublished(): returns all Tutorials with published having value as input published.  
– findByTitleContaining(): returns all Tutorials which title contains input title.

The implementation is plugged in by [Spring Data JPA](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/) automatically.

You can modify this Repository:  
– to work with Pagination, the instruction can be found at:  
[Spring Boot Pagination & Filter example | Spring JPA, Pageable](https://bezkoder.com/spring-boot-pagination-filter-jpa-pageable/)  
– or to sort/order by multiple fields with the tutorial:  
[Spring Data JPA Sort/Order by multiple Columns | Spring Boot](https://bezkoder.com/spring-data-sort-multiple-columns/)

You also find way to write Unit Test for this JPA Repository at:  
[Spring Boot Unit Test for JPA Repositiory with @DataJpaTest](https://bezkoder.com/spring-boot-unit-test-jpa-repo-datajpatest/)

**Create Spring Rest APIs Controller**

Finally, we create a controller that provides APIs for creating, retrieving, updating, deleting and finding Tutorials.

*controller/TutorialController.java*

package com.bezkoder.spring.datajpa.controller;

...

@CrossOrigin(origins = "http://localhost:8081")

@RestController

@RequestMapping("/api")

public class TutorialController {

@Autowired

TutorialRepository tutorialRepository;

@GetMapping("/tutorials")

public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required = false) String title) {

...

}

@GetMapping("/tutorials/{id}")

public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") long id) {

...

}

@PostMapping("/tutorials")

public ResponseEntity<Tutorial> createTutorial(@RequestBody Tutorial tutorial) {

...

}

@PutMapping("/tutorials/{id}")

public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") long id, @RequestBody Tutorial tutorial) {

...

}

@DeleteMapping("/tutorials/{id}")

public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") long id) {

...

}

@DeleteMapping("/tutorials")

public ResponseEntity<HttpStatus> deleteAllTutorials() {

...

}

@GetMapping("/tutorials/published")

public ResponseEntity<List<Tutorial>> findByPublished() {

...

}

}

– @CrossOrigin is for configuring allowed origins.  
– @RestController annotation is used to define a controller and to indicate that the return value of the methods should be be bound to the web response body.  
– @RequestMapping("/api") declares that all Apis’ url in the controller will start with /api.  
– We use @Autowired to inject TutorialRepository bean to local variable.

You can continue with step by step to implement this Spring Boot Server in one of the posts:  
– [Spring Boot JPA + H2](https://bezkoder.com/spring-boot-jpa-h2-example/)  
– [Spring Boot JPA + MySQL](https://bezkoder.com/spring-boot-jpa-crud-rest-api/)  
– [Spring Boot JPA + PostgreSQL](https://bezkoder.com/spring-boot-postgresql-example/)  
– [Spring Boot JPA + SQL Server](https://www.bezkoder.com/spring-boot-sql-server/)

**Run the Spring Boot Server**

Run Spring Boot application with command: mvn spring-boot:run.

**Angular 13 Front-end**

**Overview**

– The App component is a container with router-outlet. It has navbar that links to routes paths via routerLink.

– TutorialsList component gets and displays Tutorials.  
– TutorialDetails component has form for editing Tutorial’s details based on :id.  
– AddTutorial component has form for submission new Tutorial.

– These Components call TutorialService methods which use Angular HTTPClient to make HTTP requests and receive responses.

**Technology**

* Angular 13
* Angular HttpClient
* Angular Router
* Bootstrap 4

**Project Structure**

– tutorial.model.ts exports the main class model: Tutorial.  
– There are 3 components: tutorials-list, tutorial-details, add-tutorial.  
– tutorial.service has methods for sending HTTP requests to the Apis.  
– app-routing.module.ts defines routes for each component.  
– app component contains router view and navigation bar.  
– app.module.ts declares Angular components and import necessary modules.

**Setup Angular 13 Project**

Let’s open cmd and use Angular CLI to create a new Angular Project as following command:

ng new Angular13Crud

? Would you like to add Angular routing? Yes

? Which stylesheet format would you like to use? CSS

We also need to generate some Components and Services:

ng g s services/tutorial

ng g c components/add-tutorial

ng g c components/tutorial-details

ng g c components/tutorials-list

ng g class models/tutorial --type=model

**Set up App Module**

Open *app.module.ts* and import FormsModule, HttpClientModule:

...

import { FormsModule } from '@angular/forms';

import { HttpClientModule } from '@angular/common/http';

@NgModule({

declarations: [ ... ],

imports: [

...

FormsModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Define Routes for Angular AppRoutingModule**

There are 3 main routes:  
– /tutorials for tutorials-list component  
– /tutorials/:id for tutorial-details component  
– /add for add-tutorial component

*app-routing.module.ts*

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { TutorialsListComponent } from './components/tutorials-list/tutorials-list.component';

import { TutorialDetailsComponent } from './components/tutorial-details/tutorial-details.component';

import { AddTutorialComponent } from './components/add-tutorial/add-tutorial.component';

const routes: Routes = [

{ path: '', redirectTo: 'tutorials', pathMatch: 'full' },

{ path: 'tutorials', component: TutorialsListComponent },

{ path: 'tutorials/:id', component: TutorialDetailsComponent },

{ path: 'add', component: AddTutorialComponent }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

**Define Model Class**

Our main model class Tutorial will be exported in *tutorial.model.ts* with 4 fields:

* id
* title
* description
* published

**models**/*tutorial.model.ts*

export class Tutorial {

id?: any;

title?: string;

description?: string;

published?: boolean;

}

**Create Data Service**

This service will use Angular HttpClient to send HTTP requests.  
You can see that its functions includes CRUD operations and finder method.

**services**/*tutorial.service.ts*

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { Tutorial } from '../models/tutorial.model';

const baseUrl = 'http://localhost:8080/api/tutorials';

@Injectable({

providedIn: 'root'

})

export class TutorialService {

constructor(private http: HttpClient) { }

getAll(): Observable<Tutorial[]> {

return this.http.get<Tutorial[]>(baseUrl);

}

get(id: any): Observable<Tutorial> {

return this.http.get(`${baseUrl}/${id}`);

}

create(data: any): Observable<any> {

return this.http.post(baseUrl, data);

}

update(id: any, data: any): Observable<any> {

return this.http.put(`${baseUrl}/${id}`, data);

}

delete(id: any): Observable<any> {

return this.http.delete(`${baseUrl}/${id}`);

}

deleteAll(): Observable<any> {

return this.http.delete(baseUrl);

}

findByTitle(title: any): Observable<Tutorial[]> {

return this.http.get<Tutorial[]>(`${baseUrl}?title=${title}`);

}

}

**Create Angular 13 Components**

As you’ve known before, there are 3 components corresponding to 3 routes defined in AppRoutingModule.

* Add new Item Component
* List of items Component
* Item details Component

**Create Angular 13 Components**

As you’ve known before, there are 3 components corresponding to 3 routes defined in AppRoutingModule.

**Add new Item Component**

This component has a Form to submit new Tutorial with 2 fields: title & description. It calls TutorialService.create() method.

**components**/**add-tutorial**/*add-tutorial.component.ts*

import { Component, OnInit } from '@angular/core';

import { Tutorial } from 'src/app/models/tutorial.model';

import { TutorialService } from 'src/app/services/tutorial.service';

@Component({

selector: 'app-add-tutorial',

templateUrl: './add-tutorial.component.html',

styleUrls: ['./add-tutorial.component.css']

})

export class AddTutorialComponent implements OnInit {

tutorial: Tutorial = {

title: '',

description: '',

published: false

};

submitted = false;

constructor(private tutorialService: TutorialService) { }

ngOnInit(): void {

}

saveTutorial(): void {

const data = {

title: this.tutorial.title,

description: this.tutorial.description

};

this.tutorialService.create(data)

.subscribe({

next: (res) => {

console.log(res);

this.submitted = true;

},

error: (e) => console.error(e)

});

}

newTutorial(): void {

this.submitted = false;

this.tutorial = {

title: '',

description: '',

published: false

};

}

}

**components**/**add-tutorial**/*add-tutorial.component.html*

<div>

<div class="submit-form">

<div \*ngIf="!submitted">

<div class="form-group">

<label for="title">Title</label>

<input

type="text"

class="form-control"

id="title"

required

[(ngModel)]="tutorial.title"

name="title"

/>

</div>

<div class="form-group">

<label for="description">Description</label>

<input

class="form-control"

id="description"

required

[(ngModel)]="tutorial.description"

name="description"

/>

</div>

<button (click)="saveTutorial()" class="btn btn-success">Submit</button>

</div>

<div \*ngIf="submitted">

<h4>Tutorial was submitted successfully!</h4>

<button class="btn btn-success" (click)="newTutorial()">Add</button>

</div>

</div>

</div>

**components**/**add-tutorial**/*add-tutorial.component.css*

.submit-form {

max-width: 400px;

margin: auto;

}

**List of items Component**

This component calls 3 TutorialService methods:

* getAll()
* deleteAll()
* findByTitle()

It also contains a child component tutorial-details which we will create later.

**components**/**tutorials-list**/*tutorials-list.component.ts*

import { Component, OnInit } from '@angular/core';

import { Tutorial } from 'src/app/models/tutorial.model';

import { TutorialService } from 'src/app/services/tutorial.service';

@Component({

selector: 'app-tutorials-list',

templateUrl: './tutorials-list.component.html',

styleUrls: ['./tutorials-list.component.css']

})

export class TutorialsListComponent implements OnInit {

tutorials?: Tutorial[];

currentTutorial: Tutorial = {};

currentIndex = -1;

title = '';

constructor(private tutorialService: TutorialService) { }

ngOnInit(): void {

this.retrieveTutorials();

}

retrieveTutorials(): void {

this.tutorialService.getAll()

.subscribe({

next: (data) => {

this.tutorials = data;

console.log(data);

},

error: (e) => console.error(e)

});

}

refreshList(): void {

this.retrieveTutorials();

this.currentTutorial = {};

this.currentIndex = -1;

}

setActiveTutorial(tutorial: Tutorial, index: number): void {

this.currentTutorial = tutorial;

this.currentIndex = index;

}

removeAllTutorials(): void {

this.tutorialService.deleteAll()

.subscribe({

next: (res) => {

console.log(res);

this.refreshList();

},

error: (e) => console.error(e)

});

}

searchTitle(): void {

this.currentTutorial = {};

this.currentIndex = -1;

this.tutorialService.findByTitle(this.title)

.subscribe({

next: (data) => {

this.tutorials = data;

console.log(data);

},

error: (e) => console.error(e)

});

}

}

**components**/**tutorials-list**/*tutorials-list.component.html*

<div class="list row">

<div class="col-md-8">

<div class="input-group mb-3">

<input

type="text"

class="form-control"

placeholder="Search by title"

[(ngModel)]="title"

/>

<div class="input-group-append">

<button

class="btn btn-outline-secondary"

type="button"

(click)="searchTitle()"

>

Search

</button>

</div>

</div>

</div>

<div class="col-md-6">

<h4>Tutorials List</h4>

<ul class="list-group">

<li

class="list-group-item"

\*ngFor="let tutorial of tutorials; let i = index"

[class.active]="i == currentIndex"

(click)="setActiveTutorial(tutorial, i)"

>

{{ tutorial.title }}

</li>

</ul>

<button class="m-3 btn btn-sm btn-danger" (click)="removeAllTutorials()">

Remove All

</button>

</div>

<div class="col-md-6">

<app-tutorial-details

[viewMode]="true"

[currentTutorial]="currentTutorial"

></app-tutorial-details>

</div>

</div>

If you click on **Edit** button of any Tutorial, You will be directed to *Tutorial* page with url: /tutorials/:id.

This is how we embed the child component:  
– We use the child’s selector, here <app-tutorial-details>, as a directive within the parent template.  
– We use property binding to bind the viewMode and currentTutorial property in the child to the currentTutorial property of the parent.

So the tutorial-details component will look like this:

...

@Component({

selector: 'app-tutorial-details',

...

})

export class TutorialDetailsComponent implements OnInit {

@Input() viewMode = false;

@Input() currentTutorial: Tutorial = {

title: '',

description: '',

published: false

};

...

}

With @Input() decorator in the child component class, Angular passes the value for viewMode and currentTutorial to the child so that viewMode renders as true for example.

**components**/**tutorials-list**/*tutorials-list.component.css*

.list {

text-align: left;

max-width: 750px;

margin: auto;

}

You can add Pagination to this Component, just follow instruction in the post:  
[Angular Pagination example with ngx-pagination](https://www.bezkoder.com/angular-13-pagination-ngx/)

**Item details Component**

This component handles 2 things:  
– display the current selected Tutorial (from Tutorials List) if viewMode is true  
– display the form (with action buttons) for Tutorial details if viewMode is false

For getting data & update, delete the Tutorial, this component will use 3 TutorialService methods:

* get()
* update()
* delete()

**components**/**tutorial-details**/*tutorial-details.component.ts*

import { Component, Input, OnInit } from '@angular/core';

import { TutorialService } from 'src/app/services/tutorial.service';

import { ActivatedRoute, Router } from '@angular/router';

import { Tutorial } from 'src/app/models/tutorial.model';

@Component({

selector: 'app-tutorial-details',

templateUrl: './tutorial-details.component.html',

styleUrls: ['./tutorial-details.component.css']

})

export class TutorialDetailsComponent implements OnInit {

@Input() viewMode = false;

@Input() currentTutorial: Tutorial = {

title: '',

description: '',

published: false

};

message = '';

constructor(

private tutorialService: TutorialService,

private route: ActivatedRoute,

private router: Router) { }

ngOnInit(): void {

if (!this.viewMode) {

this.message = '';

this.getTutorial(this.route.snapshot.params["id"]);

}

}

getTutorial(id: string): void {

this.tutorialService.get(id)

.subscribe({

next: (data) => {

this.currentTutorial = data;

console.log(data);

},

error: (e) => console.error(e)

});

}

updatePublished(status: boolean): void {

const data = {

title: this.currentTutorial.title,

description: this.currentTutorial.description,

published: status

};

this.message = '';

this.tutorialService.update(this.currentTutorial.id, data)

.subscribe({

next: (res) => {

console.log(res);

this.currentTutorial.published = status;

this.message = res.message ? res.message : 'The status was updated successfully!';

},

error: (e) => console.error(e)

});

}

updateTutorial(): void {

this.message = '';

this.tutorialService.update(this.currentTutorial.id, this.currentTutorial)

.subscribe({

next: (res) => {

console.log(res);

this.message = res.message ? res.message : 'This tutorial was updated successfully!';

},

error: (e) => console.error(e)

});

}

deleteTutorial(): void {

this.tutorialService.delete(this.currentTutorial.id)

.subscribe({

next: (res) => {

console.log(res);

this.router.navigate(['/tutorials']);

},

error: (e) => console.error(e)

});

}

}

**components**/**tutorial-details**/*tutorial-details.component.html*

<div \*ngIf="viewMode; else editable">

<div \*ngIf="currentTutorial.id">

<h4>Tutorial</h4>

<div>

<label><strong>Title:</strong></label> {{ currentTutorial.title }}

</div>

<div>

<label><strong>Description:</strong></label>

{{ currentTutorial.description }}

</div>

<div>

<label><strong>Status:</strong></label>

{{ currentTutorial.published ? "Published" : "Pending" }}

</div>

<a

class="badge badge-warning"

routerLink="/tutorials/{{ currentTutorial.id }}"

>

Edit

</a>

</div>

<div \*ngIf="!currentTutorial">

<br />

<p>Please click on a Tutorial...</p>

</div>

</div>

<ng-template #editable>

<div \*ngIf="currentTutorial.id" class="edit-form">

<h4>Tutorial</h4>

<form>

<div class="form-group">

<label for="title">Title</label>

<input

type="text"

class="form-control"

id="title"

[(ngModel)]="currentTutorial.title"

name="title"

/>

</div>

<div class="form-group">

<label for="description">Description</label>

<input

type="text"

class="form-control"

id="description"

[(ngModel)]="currentTutorial.description"

name="description"

/>

</div>

<div class="form-group">

<label><strong>Status:</strong></label>

{{ currentTutorial.published ? "Published" : "Pending" }}

</div>

</form>

<button

class="badge badge-primary mr-2"

\*ngIf="currentTutorial.published"

(click)="updatePublished(false)"

>

UnPublish

</button>

<button

\*ngIf="!currentTutorial.published"

class="badge badge-primary mr-2"

(click)="updatePublished(true)"

>

Publish

</button>

<button class="badge badge-danger mr-2" (click)="deleteTutorial()">

Delete

</button>

<button

type="submit"

class="badge badge-success mb-2"

(click)="updateTutorial()"

>

Update

</button>

<p>{{ message }}</p>

</div>

<div \*ngIf="!currentTutorial.id">

<br />

<p>Cannot access this Tutorial...</p>

</div>

</ng-template>

**components**/**tutorial-details**/*tutorial-details.component.css*

.edit-form {

max-width: 400px;

margin: auto;

}